On the Combination of Domain Specific Modeling Languages

Antonio Vallecillo

GISUM/Atenea Research Group, Universidad de Málaga, Spain
av@lcc.uma.es

Abstract. Domain Specific Modeling Languages (DSMLs) are essential elements in Model-based Engineering. Each DSML allows capturing certain properties of the system, while abstracting other properties away. Nowadays DSMLs are mostly used in silos to solve specific problems. However, there are many occasions when multiple DSMLs need to be combined to design systems in a modular way. In this paper we discuss some scenarios of use and several mechanisms for DSML combination. We propose a general framework for combining DSMLs that subsumes them, based on the concept of viewpoint unification, and its realization using model-driven techniques.

1 Introduction

Complexity is one of the major drawbacks that UML [1] currently faces. Its metamodel of hundreds of classes and relationships between them represents a challenge for all its stakeholders. Users have serious problems for understanding its intricate structure and tend to use just the bit they know and feel comfortable with (around 20% according to the latest surveys). Formalists have problems for specifying its formal semantics and continually uncover subtle problems and ambiguities. Tool vendors find it very difficult to implement all its features (e.g., how many tools you know that can draw multiple clients or suppliers in a UML dependency?).

And even if UML provides a large number of concepts, they are still insufficient to capture some of the specific aspects required for modeling particular kinds of systems. To address this issue, UML counts on extension mechanisms for defining new modeling languages. For example, SysML [2] extends UML to define a general-purpose modeling languages for systems engineering applications. The UML Profile for MARTE [3] provides another extension of UML for modeling real-time and embedded systems. The problem, again, is the size and complexity of these extensions, which does not help making them more understandable, manageable, usable or analyzable—specially when their accidental complexity is added to the intrinsic complexity of the systems being modeled. And then we may need to combine several of these extensions, something whose results are neither clearly defined nor predictable...

The problem, as we see it, is not so much with UML itself (although it still has some issues that can be resolved, UML is a very powerful and widely used modeling notation with many supporting tools), but with its complexity—which hinders its full usability by average system modelers.

When looking for solutions, many people are starting to use Domain Specific Modeling Languages (DSMLs). These small and focused languages are becoming commonplace for specifying systems at a high-level of abstraction, using a notation very close to the problem domain and quite intuitive for the domain expert. A DSML provides a language to describe a *view* of the system, concentrating on the elements which are relevant to that particular view. However, the use of small DSMLs becomes a real problem when we need to compose them to specify a complete system. How to combine DSMLs? Which mechanisms are available for composing them? How to prove the correctness and consistency of the composition?

There is a growing number of works on DSML composition, which address the problem from different perspectives and using different combination operations: metamodel merging [1,4,5], metamodel extension [6], template instantiation [7], language embedding [8,9], different flavors of model inheritance [10], model and metamodel weaving [11,12,13] (also referred to as metamodel interfacing [7]), even product-line configuration techniques [14]. However, not all of them provide solutions to all cases, and most of them are quite limited.

In this paper we discuss different scenarios of use, and different mechanisms for DSML combination; the advantages they introduce, as well as their limitations. We propose a general framework for combining DSMLs that subsumes them, based on the concept of viewpoint unification [15] and its realization using model-driven techniques.

2 A Brief Introduction to DSMLs

When working on a large system it is unrealistic to capture all the necessary information, constraints and decisions in a single flat specification, or even in a straightforward hierarchical specification based on successive refinements [16]. Structuring the specification into viewpoints gives much more flexibility. A *view* is a representation of the whole system from the perspective of a viewpoint. Each view focuses on the elements relevant to that particular viewpoint, abstracting away all irrelevant details. The view elements represent the system elements, as seen from the corresponding viewpoint.

Each viewpoint has a *viewpoint language* (i.e., a DSML) for describing the corresponding views. Each view then is a *model* that conforms to the corresponding DSML metamodel. Because the different viewpoints stress different aspects of the design, and use different techniques for doing so, each designer (or stakeholder) will be most comfortable with their own style of language and notation. For example, people writing processes and algorithms will probably think better in *imperative* terms (and use xUML, BPMN or Java), while business rule experts will find more suitable a *declarative* language (such as SVBR or OCL). Moreover, the models describing the separate views are independently expressed: they are each formed from a separate set of interrelated concepts, but no model element makes direct reference to terms in any other view model.

The goal of DSMLs is to allow domain experts to specify and reason about their systems using intuitive notations, closer to the language of the problem domain, and at the right level of abstraction. These are *specific* because they restrict themselves to one particular problem domain, supporting higher-level abstractions than general-purpose modeling languages and sacrificing generality to gain in specificity and concreteness.

This makes them easy to learn and to use (by the domain experts), manageable, usable and analyzable. Furthermore, the rules of the domain are included into the language as constraints, disallowing the specification of illegal or incorrect models of the views.

Finally, we should recall that defining a DSML involves at least three aspects: the domain concepts and rules (abstract syntax); the notation used to represent these concepts—let it be textual or graphical (concrete syntax); and the semantics of the language. The *abstract syntax* of a DSML is normally defined by a metamodel, which describes the concepts of the language, the relationships between them, and the structuring rules that constrain the model elements and their combinations in order to respect the domain rules. The *concrete syntax* of a DSML provides a realization of the abstract syntax of a metamodel as a mapping between the metamodel concepts and their textual or graphical representation. A language can have several concrete syntaxes. Finally, a DSML may have different kinds of *semantics*, depending on the aspects we want to emphasize. Thus, we can have structural semantics (describing what correct models produced with this DSML actually mean), behavioral semantics (how they behave along some time model), etc. [17]

3 Mechanisms for Combining DSMLs

The fact that each view provides only a partial specification for the system, requires mechanisms for combining DSMLs (and also their corresponding models) to be put in place. It is essential to observe that the combination of DSMLs should yield another Modeling Language (although not "Domain Specific" any more!), able to represent a metamodel for the "unified" models that provide a reconciled, integrated and virtual representations of the separate views of a system specification.

The following questions need to be answered: How can such a combined Modeling Language be built? How does it relate back to the individual DSMLs (and associated tools)? How to construct its metamodel? And its concrete syntax? How to define its semantics? These are the questions that we will try to answer here.

Note that such a combined Modeling Language (and its associated metamodel) can become too complex to be usable by modelers, and will not normally be presented to any user. Same as it happens with the output of a program compiler, which produces an executable model by combining information about the program itself, the execution platform, the hardware architecture, etc. The resulting model, which is in binary form, is not for human consumption; users only deal with specific views of it: the functionality, the configuration files, the information about the dynamic libraries, the deployment information, etc. Compiler and associated tools make the appropriate connections. Other tools, such as symbolic debuggers, can use parts of these models to provide the user with *new* views of the system at a high level, for instance during program execution.

The final goal is that tools can construct part or all of such a unified model where they need to manipulate information from more than one viewpoint, or to extract information from it. In this way, the user will normally work with the individual DSMLs, and leave the combining tools to build the unified models as needed.

For the combination of two or more DSMLs (and their associated metamodels and models) we need to address three main issues: (1) how to describe the correspondences between the concepts of the languages (i.e., at the metamodel level) and between the elements in each view (i.e., at model level); (2) how to "integrate" the models that represent the views into a global workable model (using the views and their correspondences); and (3) how to relate the unified model with the original views, so that the original views can be extracted from the unified model. The first problem deals with **Relating** the individual views; the second one with their **Synthesis**; and the last one with the **Analysis** of the unified model.

3.1 Relating Models: Correspondences

Dividing a system specification into a set of views provides a powerful mechanism for achieving the required level of abstraction, simplicity and modularity. However, the specifications must be a *coherent* description of a single target system. It is therefore essential that the views be linked, and this is done by establishing a set of **correspondences** between them. Correspondences do not form part of any one of the DSMLs, but provide statements that relate the various different views—expressing their semantic relationships [16]. Hence, a proper system specification consists of a set of viewpoint specifications, each one expressed in a viewpoint DSML, together with a set of correspondences between them.

The majority of the existing proposals for viewpoint modeling do not consider correspondences between viewpoints, or assume they are trivially based on name equality between correspondent elements, and implicitly defined. In fact, most proposals and tools for merging models (including UML 2) take a simplistic approach to matching based on names: if the same name appears in two views, they are assumed to represent two aspects of the same object. However, if the models are to be developed by separate teams, it is not safe to assume they share a single namespace, or that name assignments are unique. It is also often the case that the correspondences are not simply one-to-one; the relationships between elements will generally be more complex.

Several authors have proposed different approaches to express correspondences, specially when views are expressed as UML models, using different alternatives: from OCL constraints to UML abstraction dependencies (see [18] for a discussion about some of these approaches). Other proposals use model weaving techniques for relating the elements of different views, defining ad-hoc correspondence metamodels [19], general-purpose model weaving notations and tools [12,13] or even bi-directional model transformation languages such as QVT.

Correspondences need normally be specified at two levels, depending on whether they relate metamodel or model elements. In the first case, correspondences determine the relationships that should exist between concepts of the two DSMLs to be combined. For example, if we are combining class diagrams with statecharts, a correspondence between the two language metamodels can specify that every UML class should be related to one or more statecharts (the ones that define the behavior of the instances of that class). But then, instances of such correspondences (called *correspondence links*) should be specified at the model level, identifying which are the individual statecharts that should be related to a particular class. Making an analogy with programming languages, you need to define first how the grammars of the two languages can be related, and then how two individual programs are related using such relations.

There are also situations where establishing correspondences becomes a difficult task, and cannot be automated. For example, a complex structure in one model can express a concept that is expressed by another complex structure in another model, but there is no obvious mapping for the individual elements even though the structures as a whole are similar. Correspondences between non-structural elements (e.g., constraints or pieces of behavior) are not trivial, either. A very illustrative introduction to the nature of correspondences and their associated problems and limitations can be found in [16].

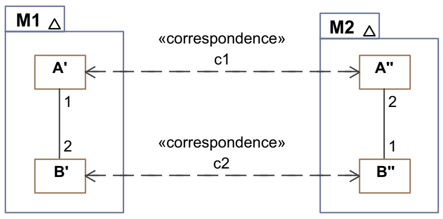

Fig. 1. Correspondences between two models

It is also worth noting that, given two models, there are many different ways of relating their elements. An example of the correspondence between two models (that may represent different viewpoints, or different views) is depicted in Fig. 1. Depending on the constraints defined by the individual correspondences, the specifications can be "consistent" or not. The key question here is the meaning of *consistency*. We will come back to this later in Sect. 4. So far we would only like the reader to consider if the system specification shown in Fig. 1 is consistent w.r.t. the two defined correspondences (`c1` and `c2`) or not.

3.2 Viewpoint Synthesis

Some authors have proposed a number of techniques for combining (meta)models. They can be basically grouped in three categories, which are discussed in this section.

Metamodel Extension. One possible approach to DSML combination consists of extending one language (the pivot) with the concepts of the other (the extension). These new elements were not originally present, but some of them may make references to existing ones. There are several situations where such an extensibility mechanism is useful and essential, e.g., in the case of hierarchies of metamodels or to modularly endow a language with features not originally present.

An ***extensionOf*** relation between the two language metamodels was formally introduced in [6]. It subsumes previous proposals for implementing different flavors of *model inheritance* [10] or *template instantiation* [7].

Given two metamodels[1] M_i and M_e that conform to the same reference model (or meta-metamodel) $\mathcal{M}$ and that represent the initial metamodel and the extension (M_e), and given a correspondence mapping $\epsilon : M_i \longrightarrow M_i \cup M_e$ that defines how elements in the initial metamodel are mapped to elements in the union model (the one that contains all elements of both metamodels), the authors in [6] show how to compute the **synthesized** metamodel $M_i \oplus_\epsilon M_e$ with the "duplicate-free union" of the two metamodels being combined.

Here, the relationship between the metamodels is accomplished by a user-defined specification ϵ of the correspondences between the elements that should be "unified".

This approach to DSML composition is effective when we want to re-use an existing DSML and complement it with another (that can be reused, too), and the relation

[1] Metamodels are models too, so most of the definitions of this paper apply equally to models.

between the two is *complementary* (the extending language *complements* the other) and *conservative* (the extensions are compatible with the pivot language's concepts and do not break its semantics). Aspect-oriented modeling approaches could fit into this category, since they allow extending models with new properties. Other languages provide extension mechanisms for facilitating this task (e.g., UML Profiles allow extending the UML metamodel to incorporate new features).

Another benefit of this approach is that the concrete syntax of the resulting DSML can be easily defined (see, e.g., [20]), and the combined semantics can be defined as well (at least in theory), because the extensions have to be conservative [4].

One disadvantage of this approach is its limited use, only for conservative extensions of a language and not for combining DSMLs in general. In addition, combining separate extensions is not a trivial task: although each one can be conservative w.r.t. the pivot language, the consistency of the extensions compositions is not guaranteed (two extensions may impose contradictory conditions on the global combined model).

Metamodel Merge. Model merge is a more powerful composition operation that does not assume an unbalanced combination, but tries to combine *peer* languages. For example, UML 2 defines an operation, package merge, that takes the contents of two packages (models or metamodels) and produces a new package that combines their contents [1]. Package merge was partially inspired by two specification combination mechanisms offered in Catalysis: "and" and "join" [4]. However, both differ substantially from package merge: The "and" operation is for use with subtyping, while the "join" operation allows a specification to impose additional preconditions to those defined in another view [21]. The problem, as it stands today, is that the current definition of this UML operation is neither precise nor sound, and it does not consider possible conflicts between the structural constraints of the metamodels that are merged. As a result, it may break the well-formed rules of any of the languages it combines [4]. Besides, the solution adopted in UML 2 is too simplistic: elements are merged based on name matching and the resulting extended elements have all the properties of the elements they merge (we shall see that this becomes a problem, too).

MetaGME [7] enables Metamodel Merge through the use of three types of class inheritance and a special Class Equivalence operator, used to show a full union between two classes. The unioned classes cease to exist as distinct metamodel elements, instead fusing into a single class. The union process is very similar to merging classes through Package Merge, except that the operation takes place at the class level instead of the package (or metamodel) level, and the two merged classes do not need to have the same name because of the use of the Class Equivalence operator.

Pottinger and Bernstein proposed in [5] a more general approach to model merging, using user-defined correspondences between the views. They presented an algorithm that, given the two models and a set of user-defined correspondences between them, provides a merged model which is the duplicate-free union of the two models with respect to the set of correspondences. The authors identify different kinds of possible conflicts, some of which may be resolvable, others are not in general. Their approach subsumes previous works from the database and semantic web communities on generic model merging, database view integration and ontology merging, by generalizing these approaches and providing a unified algorithm.

Although trying to compute the duplicate-free union of two metamodels by merging them could a priori be an excellent solution for DSML combination, it does not work in all cases. Merges have to be meaningful from an architectural (and methodological) point of view: not always the metamodels of two languages are amenable to merging because their underlying semantics are different and incompatible. Think for instance of two languages for describing behavior, one based on synchronous interactions and the other on asynchronous interactions. You can relatively easily relate their metamodels using correspondences, but you cannot easily merge them into a single unified metamodel. A similar situation happens if you try to merge a Class and E/R notations into one single unified language. Or think of combining Java and COBOL programs into the same language. Or programs written in my two favorite DSLs: $\LaTeX$ and Excel...

Furthermore, merging models usually implies carrying forward all the properties of all merged model elements. In other words, model merge only allows injection relationships between the models being merged and the resulting model. For example, in UML an element resulting from the merge must not be any less capable than it was prior to the merge. This means, among other things, that the resulting navigability, multiplicity, visibility, etc. of a receiving model element will not be reduced as a result of a package merge [1]. Then, if you consider again the models in Fig. 1, merging the classes according to the correspondences leads to inconsistent cardinality constraints. Does this mean that these two models cannot represent views of the same system? Probably they can (see, e.g., Fig. 3, whose orthographic views M1 and M2 have the same constraints), but the problem is that package (or model) merge is not the right combination operator for integrating them.

Language Embedding. An alternative approach to building a DSML from scratch is to inherit the infrastructure of some other language, tailoring it in special ways to the domain of interest. This is called language embedding [8,9]. In this way, the embedded language can reuse the syntax of the host language, its module system, existing libraries, associated tools, etc. The embedding is normally defined in terms of a mapping function that describes how the guest language concepts are encoded in terms of the host language concepts. Furthermore, in case of host languages with precise semantics, the embedding mapping can serve to provide translational semantics to the guest language (i.e., the semantics of the guest language concepts is defined in terms of the interpretation of the translated concepts in the host language).

In other words, if MM_g and MM_h are the language metamodels of the guest and host languages, the embedding is a mapping $\varepsilon : MM_g \rightarrow MM_h$. Normally, such a mapping is not explicitly defined anywhere, and there is no explicit trace between the two languages—losing therefore the connection with the concrete syntax and tools associated to the original DSML.

Of course, the host language should be expressive (and malleable) enough to represent the concepts of the guest. Usually, functional languages such as Haskell or Scala, or formal-based languages such as Maude have proved to be good hosts.

UML has been used as the host language for a wide range of DSMLs. UML is very expressive, well-know and it counts on tool support—well, mainly model editors. In fact, UML was originally created to combine (by hosting) the original Booch, OMT and OOSE methods and notations, incorporating slightly modified versions of languages such as Harel's Statechart notation [22], or ITU-T's Message Sequence Charts (MSC) [23]. Thus, UML defines a global metamodel with all the original notations it combines (for use cases, class diagrams, state charts, sequence charts, etc.).

From a theoretical perspective, the use of the host language metamodel can help maintaining the coherence and conceptual integration among the viewpoints elements. However, this approach presents some problems from a practical point of view. Firstly, in many occasions it means re-defining the original languages to integrate them into the host language metamodel, something which normally hampers the use of existing editors and analysis tools for the original languages (e.g., the tools available for Harel's Statecharts or for ITU-T's Message Sequence Charts are not easily accessible from UML). Secondly, some of the adaptations have respected the original semantics of the languages, but others had to suffer some modifications or severe cuts (e.g., Statecharts in UML 1). Thirdly, the relationship between the elements of the different languages is not obvious in general, and gets usually blurred—mainly because of the intricate nature of the global metamodel, and because in many cases it is built without mechanisms for expressing the correspondences between the viewpoints. Finally, language embedding may force to ask users to stop using their domain specific notations, small and concise languages and specific tools, and to start using a (probably more) complex language (at least, far more expressive).

In general, a common Modeling Language that accommodates all DSMLs is feasible if the number of viewpoints is small and semantically consistent, and if as user you are happy to forget about the individual DSMLs and their associated tools. But it is rather artificial if the DSMLs are loosely coupled or describe the system at very different levels of abstraction/granularity.

Embedding and extensions. In many occasions, host languages also count on extension mechanisms for facilitating the embeddings[2]. For example, UML counts on Profiles to help defining/hosting new languages. UML Profiles also allow users to define the embedding function explicitly, indicating which UML metaclasses are extended.

Another example is WebDSL [24], a textual DSML for developing dynamic Web Applications that incorporates different languages for expressing the concerns involved in any Web system. WebDSL is is extensible, so new languages can be added as plugins to cope with new concerns. No explicit embedding mappings ε between the guests and host language are defined, though.

Embedding languages in this way is not free from problems, either. Let us mention the most significant issues that we have found when working with UML profiles, although they are generic to this kind of approaches. First, well-defined UML profiles cannot break the semantics of UML (at least, in theory); however, they can easily introduce semantic inconsistencies between each other when two or more, independently defined, are applied together (e.g., see the problems of combining SysML and MARTE profiles in [25]). Second, the use of UML as a modeling notation introduces some restrictions and limitations, which may force design choices sometimes unnatural when modeling certain domain concepts; for example, SysML models *Requirements* by

[2] In these cases, extending a language can also be seen as a form of embedding. The difference is usually a matter of degree, and from where we look at it: from the host side (that gets extended) or from the guest side (that gets embedded).

extending UML classifiers, a decision which can be considered (at least) arguable. Finally, the complexity of the UML metamodel does not help when looking for elements that can represent the domain concepts. In our previous experience with UML profiles to model languages such as WebML [26] or the RM-ODP viewpoint languages [19], we found that some times we had too many choices (e.g., it was difficult to decide whether some concepts had to be represented by UML classes or by UML components, because their differences are quite subtle), while in other occasions we could not find any UML element to represent what we wanted (e.g., expressing ODP policies was not a trivial task). There is also the issue of the concrete syntax: adopting UML graphical notation is a suitable choice when the embedded language does not have its own concrete syntax (such as UML4ODP or SysML) because many people are familiarized with UML boxes and lines, and the learning curve is small; but the results obtained when trying to mimic other concrete syntaxes are not good, basically due to the reduced facilities of UML Profiles for adopting new graphical notations [26]. Worse than that, what we have found is a recurrent undesirable situation when modelers embed DSMLs into UML. Since the frontier between the embedded and the host language disappears, users start making use of many UML concepts that were not part of their original DSMLs, producing models that are correct w.r.t. the UML metamodel, but incorrect w.r.t. their original languages.

The key question, as we mentioned at the beginning, is whether users should know about this combined Modeling Language at all, or should the tools be responsible for converting the models written in the original DSMLs, back and forth to the integrated model written in the global language. In this way, the user will normally work with the individual DSMLs, and leave the combining tools to build the unified models as needed. Probably in this scenario is where the full potential of UML could be better exploited.

3.3 Analysis of the Integrated Models

Independently from how the synthesized model has been built, there should be a way to extract the views from the integrated model. Although not so much discussed in the MDE community, this is a well known problem in databases, a part of the *data integration* problem [27]. This is the problem of combining data residing at different sources, and providing the user with a unified view of the data. In this approach, the user queries over the global schema have to be reformulated in terms of a set of queries over the sources.

One of the current limitations of language embedding is that there is no trace back to the original language that has been embedded. Basically, more than "combining" the languages, they are re-defined from scratch using the metamodel of the host language, and with no explicit backward connections to the original DSMLs. This is for instance the case of UML with statecharts or MSCs. The situation is not better with languages defined using UML Profiles: although the embedding mapping ε is explicitly defined, the reverse projection is not.

Furthermore, what happens with the tools (editors, analyzers, etc.) of the individual views? It is important to have access to the tools available for individual DSMLs from the combined DSML environment. It is not clear how this can be achieved using language embedding mechanisms. This is another reason for being the tools, and not the users, the ones that should combine their models into an integrated Modeling Language.

4 Viewpoint Unification

Our proposal for DSML combination builds on the idea of viewpoint unification, originally proposed by Boiten, Derrick, Bowman and Steen for studying the consistency between viewpoint-based specifications [15]. In that work, a set of viewpoints is considered to be *consistent* if there exists at least one "implementation" that satisfies all the views. This is equivalent to check that the views do not impose contradictory requirements on the system. A detailed study of the formal basis for viewpoint unification mechanisms can be found in [15]. Here we extend that notion in order to deal as well with the correspondences between the viewpoints, and with the explicit representation of the relations between the unified model and the views.

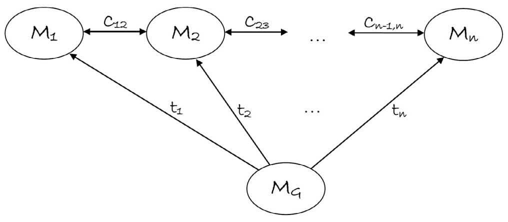

Fig. 2. A unified model

The idea consists in considering that the DSMLs to combine provide a set of *viewpoint languages* to describe one system, and hence the models written according to these DSMLs represent the system *views* (as proposed in RM-ODP [28] or in [29]). Because all viewpoints correspond to the same system, and will eventually be realized by one implementation, there must be a way to combine them. Intuitively, the way to combine the languages is by providing a *new language* and a set of *mappings* between the new language and the viewpoint languages (Fig. 2), with the additional property that the mappings respect the constraints imposed by the correspondences.

The more general process to create the metamodel of the new language M_G and the mappings $t_1, \dots, t_n$ is based on the **unification** of the viewpoint languages metamodels. The mappings capture the relations between the unified metamodel and the individual viewpoints metamodels, acting as **projections** of M_G [29]. The consistency of the specification is guaranteed by the fact that the mappings should respect the correspondences between the viewpoints: two projections of the same system over two different viewpoints must be related by the correspondences in a consistent way.

Definition 1 (Model Unification). *Given a set of models $M_1, M_2, \dots, M_n$, and a set of correspondences between them $c_{ij} = C(M_i, M_j) \subseteq \mathbb{P}(M_i) \times \mathbb{P}(M_j)$, a* unification *is a new model M_G and a set of functions $t_i : M_G \to M_i$ (*projections*) that respect the set correspondences, i.e., $C(t_i(M_G), t_j(M_G))) \subseteq C(M_i, M_j)$.*

In case of combining DSMLs by unification, models $M_1, M_2, \dots, M_n$ are the metamodels of the languages to combine, and M_G is the metamodel of the unified language.

The form of unification depends on the DSMLs to be combined, the correspondences defined between them, and the different relations that can be defined between the unification and the views. For example, the metamodel M_G of the unified language could be defined by applying model extension or model merge operations on the metamodels of the viewpoint languages (in those cases where this makes sense). Or we could use the metamodel of an existing language as global metamodel M_G (this is language embedding). Alternatively, unification offers further options such as defining an ad-hoc metamodel (neither the duplicate-free union nor an existing language metamodel) for combining particular DSMLs, as we shall see below.

We can also identify different kinds of mappings, depending on the sort of relationship between the unified metamodel and each viewpoint metamodel—we should allow to relate them in different ways. In some proposals, the mappings are defined between the viewpoint languages and the unified model, and they are called *development* relations [15]. They represent the inverse mappings of our projections. For instance we can have *refinement* relations, *abstractions*, *equivalences* and relations which can broadly be classified as *implementations*. These different kinds of relations are best distinguished by their basic properties. Refinements are reflexive and transitive (i.e., a preorder); abstractions are the dual of refinements; equivalences are reflexive, symmetric and transitive; and implementation relations only need to be reflexive [15]. Transitivity is a very expensive property, but crucial for enabling incremental development of specifications towards realizations. Implementation relations are the most common relations, they just establish correspondences between the unified metamodel and the viewpoint metamodels. For example, consider a requirements specification of the system written using OMG's Business Motivation Model (BMM) notation and a functional specification using LOTOS (ISO/IEC 880). A unified model may be expressed in a completely different notation, and related to the former by a logical satisfaction relation, and to the latter by a behavioral conformance relation.

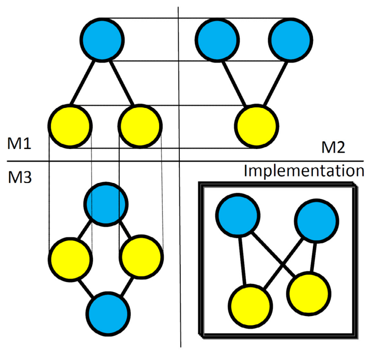

Fig. 3. Orthographic views of a 3D object

Our approach to DSML combination subsumes previous approaches (see Sect. 5), and allows a wider range of possible combinations. For example, consider again the models in Fig. 1. Merging them was not possible because the merge operator finds inconsistencies between the cardinality constraints of the classes to merge. However, consider the orthographic representation of a 3D object shown in Fig. 3, whose views M1 and M2 present similar correspondences to the classes in Fig. 1, but for which a combined model is possible (shown as the Implementation).

In fact, the two models shown in Fig. 1 admit one unification, given by a model MG with two classes A and B related by an association whose cardinality is 2 in both ends, plus two projections T1 and T2—see Fig. 4.

The first projection T1 transforms each pair of A instances of MG related to a pair of B instances into one single A' instance, and transforms B instances into B' without modifying them. The second projection T2 does the analogous transformation with B" and A" instances, respectively. This represents, for instance, a system in which both A and B elements are replicated. View M1 abstracts away the replication of A elements, while View M2 does the same for B elements.

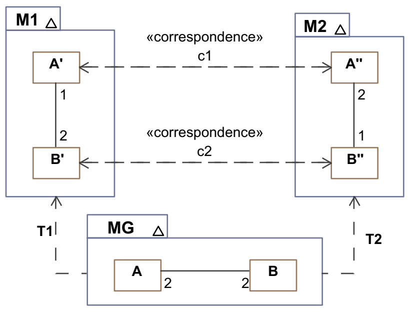

Fig. 4. The Unified Model for Fig. 1

Of course, the unification is only possible if correspondences c1 and c2 are one-to-many and many-to-one, respectively. Otherwise the unification is not possible: suppose that correspondence c1 was one-to-one, i.e., it imposed that every A' instance should be related to exactly one A" instance. In this case, there is no implementation possible for the system and therefore the multi-viewpoint specification becomes *inconsistent*.

5 Discussion

Relationship with previous approaches. Our proposed unification can be seen as a generalization of previous approaches to DSML combination, discussed in Sect. 3. In model extension and model merge, the unified metamodel M_G is nothing but the duplicate-free union of the viewpoint metamodels. The *development* relations in these cases coincide with injection mappings defined by these two approaches (e.g., ϵ in case of model extension), and the projections t_i are just the inverse of these injections. One of the benefits of our approach is that we request that the projection mappings are explicitly defined. One of the benefits of the model extension and model merge approaches is that they provide mechanisms and algorithms for building the unified model (in those situations in which they can be applied), because the unified model coincides with the combined model they construct—sometimes called the *least developed unification* [15].

Language embedding is also a particular case of our approach, in which the metamodel M_G is an existing one. Users normally define the development mappings that describe how the DSMLs concepts are encoded as M_G elements. In our case, we explicitly ask to specify the inverse projections t_i too, to be able to trace back to the original languages and to automatically obtain the views from the unified model.

We are of course conscious that the synthesis process cannot always be fully automated, as we have tried to illustrate with the simplistic example shown in 4. There are, however, other occasions in which such a combined model can be synthesized from the views, as it happens when model extension or model merge approaches are possible. But in these cases the projections are easy to define, because they are nothing but the inverse of the development mappings.

Realizing the Mappings. The advent of MDE has provided a set of appropriate mechanisms and tools for specifying and implementing both the viewpoint correspondences and the t_i projections. For instance, model weaving [12] is a technology that can be very useful to implement model extension and model merging, as described in [11,13]. More importantly, model transformations can play a key role for realizing the mappings so that they can be automated. In the best case we will be able to define bidirectional model transformations that allow the mappings to work in both directions.

In a typical application scenario, a user will be confronted with two DSMLs that have to be combined. The first step is to define the correspondences between their metamodels using model weaving techniques. Then the user should investigate whether model extension or model merge can produce a satisfactory and consistent unified metamodel (in case the projections of the duplicated-free union of the two languages metamodels respect the correspondences constraints). If so, implementing the algorithms described in [6] or in [5] using model transformations is the solution. Once defined, the projections from the unified metamodel to the views should be defined in terms of model transformations, to be able to perform automatic analysis (these projections are basically the inverse of the development mappings defined by the algorithms).

Alternatively, the user may consider embedding the languages into an existing language, if none of the issues we have identified in Sect. 3.2 represent a serious problem for her. In this case the development relations are just the embedding mappings, which can be implemented in terms of model transformations, too (see, e.g., [30,31,32,33]). Apart from the intrinsic problems of defining the mappings and the projections (which are not normally difficult from a conceptual point of view but rather cumbersome from a technical perspective), special care should be taken for making sure that the correspondences constraints are respected by the projections.

Finally, in case none of the previous approaches offers a neat solution, the user might consider specifying an ad-hoc language for hosting the combination. As major benefits, the relationships between the combined DSMLs and the unified language can be of different types, and implemented as model transformations (in both directions: development and projections) that will fit the particular requirements of the individual languages. The main problem is the complexity involved in defining the unified language so that it represents the consistent "least development unification" of the DSMLs to combine. The good news is that this new language has to be defined only once for every combination of languages.

What happens with the concrete syntax? In our proposal, users do not need to use the combined language and thus there is no need to provide a concrete syntax for it. In case of model extension, some authors have proposed a way to combine the concrete syntax as well [20]. But in general this is a difficult issue because of the semantic implications of symbols: usually every symbol conveys an associated meaning. For instance, a box is associated to a classifier in UML; a stick figure is an actor, etc. There is no major problem when the concepts of the combined languages are kept separated, or just extended, but not mixed. But when the concepts are mingled in the combined language the situation becomes more complex, and trying to use the icons of one or the other language may introduce semantic problems to the reader of the combined diagrams. And if we try to choose a different notation for the combined language, the users might get completely confused with the new notation.

In language embedding we get the opposite problem because users tend to focus more on the host language notation—the embedded language symbols usually become (inconized) annotations to the host language symbols. But the look-and-feel of the resulting diagrams resembles too much the host language notation, and thus the benefits of working with *domain specific* languages melt away. However, this might not be a problem but an advantage when the embedded language does not have any associated concrete syntax, as we explained before.

What happens with the semantics? There have been different proposals for the compositional definition of the semantics of DSMLs using diverse formalisms, see, e.g., Refs. [34,35,36,37]. These works are usually valid when the relation between the viewpoints and the unified metamodel are basically injections. But in general combining the semantics of the languages is not a trivial task and deserves its own line of research—specially when we allow different kinds of relations between the unified metamodel and each viewpoint metamodel. In an unification context, the semantics of the individual DSMLs and the unified language are preserved. Model transformations provide here the *semantic brigdes* that allow mapping ones into the others. Furthermore, model transformations can serve to define the (translational) semantics of those languages that do not count on an explicit definition of their semantics, as mentioned in Sect. 3.2.

6 Conclusions and Future Work

In this paper we have discussed and analyzed the most common techniques for DSML combination, and classified them in three main categories according to the operations they use: model extension, model merge and language embedding. These techniques are useful in some circumstances, but rather limited in others. Then we have proposed a more general framework for combining DSMLs that subsumes them, based on the concept of viewpoint unification, and its realization using model-driven techniques. The framework has allowed us to put these combination techniques in context, and formulate them in similar terms. In fact, they all represent different ways to find a global metamodel that can host the languages to combine. But these approaches have similar problems, too. Firstly, none of them specifies in an explicit way the traces back to the original notations that permit making use of the tools available for these languages. Secondly, they allow only one kind of relationship between the languages to combine and the global metamodel (basically, injection). The first problem is solved in our proposal by requesting the explicit specification of the mappings from the global metamodel to the languages metamodels. The second problem is the one that imposes stronger limitations on existing approaches for combining DSMLs because it forces the global metamodel elements to incorporate all the capabilities of the individual views, and to respect the constraints defined by both the viewpoints and the correspondences. We have introduced a simple example that shows that such limitation is too restrictive, and normally unrealistic for composing rich DSMLs. Our approach overcomes this limitation by allowing different kinds of relations between the viewpoint languages and the global metamodel (abstractions, refinements, implementations, etc.) and also by checking the consistency of the specifications using the projections of the global metamodel.

We are currently working on the unification of the viewpoint languages defined by some multi-view proposals, such as UWE [38] and the RM-ODP [28]. This is the context in which the work presented here has been developed, based on our experiences and findings when combining these languages. Although there are still many issues to resolve, we have tried to show how the MDE technologies can significantly help in combining DSMLs by formulating the problem in terms models and relations (transformations) between them.

Acknowledgements. We would like to thank José E. Rivera, Francisco Durán and Jordi Cabot for their comments on a previous version of this paper, and to the anonymous referees for their insightful comments and suggestions. This work has been partially supported by Spanish Research Projects TIN2008-03107 and P07-TIC-03184.

References

1. OMG: Unified Modeling Language 2.1.1 Superstructure Specification. OMG, Needham (MA), USA, OMG doc. formal/07-02-05 (2007)
2. OMG: Systems Modeling Language. OMG, Needham (MA), USA (2008)
3. OMG: UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded systems. OMG, Needham (MA), USA (2009)
4. Zito, A., Diskin, Z., Dingel, J.: Package merge in UML 2: Practice vs. theory? In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 185–199. Springer, Heidelberg (2006)
5. Bernstein, P.A., Pottinger, R.A.: Merging models based on given correspondences. In: VLDB 2003, Berlin, Germany pp. 862–873 (2003)
6. Barbero, M., Jouault, F., Gray, J., Bézivin, J.: A practical approach to model extension. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 32–42. Springer, Heidelberg (2007)
7. Emerson, M., Sztipanovits, J.: Techniques for metamodel composition. In: Proc. of the 6th Workshop on Domain Specific Modeling at OOPSLA 2006, pp. 123–139 (2006)
8. Hudak, P.: Building domain-specific embedded languages. ACM Comput. Surv. 28(4) (1996)
9. Hofer, C., Ostermann, K., Rendel, T., Moors, A.: Polymorphic embedding of DSLs. In: Proc. of GPCE 2008, Nashville, TN, pp. 137–148. ACM, New York (2008)
10. Ledeczi, A., Nordstrom, G., Karsai, G., Volgyesi, P., Maroti, M.: On metamodel composition. In: Proc. of CCA 2001, pp. 756–760 (2001)
11. Estublier, J., Vega, G., Ionita, A.D.: Composing domain-specific languages for wide-scope software engineering applications. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 69–83. Springer, Heidelberg (2005)
12. Didonet Del Fabro, M., Jouault, F.: Model transformation and weaving in the AMMA platform. In: GTTSE 2005. LNCS, vol. 4143, pp. 71–77. Springer, Heidelberg (2005)
13. Bézivin, J., Bouzitouna, S., Didonet Del Fabro, M., Gervais, M.P., Jouault, F., Kolovos, D., Kurtev, I., Paige, R.F.: A canonical scheme for model composition. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 346–360. Springer, Heidelberg (2006)
14. White, J., et al.: Improving domain-specific language reuse with software product line techniques. IEEE Software 26(4), 47–53 (2009)
15. Bowman, H., Steen, M., Boiten, E.A., Derrick, J.: A formal framework for viewpoint consistency. Formal Methods in System Design 21(2), 111–166 (2002)
16. Linington, P.: Black Cats and Coloured Birds What do Viewpoint Correspondences Do? In: Proc. of WODPEC 2007, Maryland, USA (2007)
17. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling, 2nd edn., Ceteva (2004)

18. Romero, J.R., Jaén, J.I., Vallecillo, A.: Realizing correspondences in multi-viewpoint specifications. In: Proc. of EDOC 2009, Auckland, NZ, pp. 163–172. IEEE Computer Society, Los Alamitos (2009)
19. ISO/IEC: Information technology – Open distributed processing – Use of UML for ODP system specifications. ISO and ITU-T, ISO/IEC IS 19793, ITU-T X.906 (2008)
20. Pedro, L., Risoldi, M., Buchs, D., Barroca, B., Amaral, V.: Composing visual syntax for domain specific languages. In: Proc. of HCI 2009, San Diego, CA. LNCS, vol. 5611, pp. 889–898. Springer, Heidelberg (2009)
21. D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML. In: The Catalysis Approach, Addison-Wesley, Reading (1999)
22. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming 8, 231–274 (1987)
23. ITU-T Recommendation Z.120: Message Sequence Charts (1994)
24. Groenewegen, D.M., Hemel, Z., Kats, L.C.L., Visser, E.: WebDSL: A Domain-Specific Language for Dynamic Web Applications. In: Mielke, N., Zimmermann, O. (eds.) Companion to OOPSLA 2008, pp. 779–780. ACM, New York (2008), `http://webdsl.org`
25. Espinoza, H., Cancila, D., Selic, B., Gérard, S.: A practical approach to model extension. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 98–113. Springer, Heidelberg (2009)
26. Moreno, N., Fraternali, P., Vallecillo, A.: WebML Modelling in UML. IET Software 1(3), 67–80 (2007)
27. Lenzerini, M.: Data integration: A theoretical perspective. In: Proc. of PODS 2002, pp. 233–246 (2002)
28. ISO/IEC: RM-ODP. Reference Model for Open Distributed Processing. ISO and ITU-T, Geneva, Switzerland, ISO/IEC 10746, ITU-T Rec. X.901-X.904 (1997)
29. Atkinson, C., Stoll, D.: Orthographic modeling environment. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 93–96. Springer, Heidelberg (2008)
30. Abouzahra, A., Bézivin, J., Didonet Del Fabro, M., Jouault, F.: A practical approach to bridging domain specific languages with UML profiles. In: Best Practices for Model Driven Software Development Workshop at OOPSLA (2005)
31. Bézivin, J., Hillairet, G., Jouault, F., Kurtev, I., Piers, W.: Bridging the MS/DSL tools and the Eclipse modeling framework. In: Proc. of the International Workshop on Software Factories at OOPSLA (2005)
32. Wimmer, M., Schauerhuber, A., Strommer, M., Schwinger, W., Kappel, G.: A semi-automatic approach for bridging DSLs with UML. In: Proc. of 7th Workshop on Domain-Specific Modeling at OOPSLA (2007)
33. Brambilla, M., Fraternali, P., Tisi, M.: A transformation framework to bridge Domain Specific Languages to MDA. In: Chaudron, M.R.V. (ed.) Models in Software Engineering. LNCS, vol. 5421, pp. 167–180. Springer, Heidelberg (2009)
34. Chen, K.: et al.: Semantic anchoring with model transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 115–129. Springer, Heidelberg (2005)
35. Ruscio, D.D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for supporting dynamic semantics specifications of DSLs. Technical Report 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA), Nantes, France (2006)
36. Doh, K.G., Mosses, P.D.: Composing programming languages by combining action-semantics modules. Sci. Comput. Program. 47(1), 3–36 (2003)
37. Pedro, L., Amaral, V., Buchs, D.: Foundations for a Domain Specific Modeling Language prototyping environment: A compositional approach. In: Proc. of the DSM workshop at OOPSLA 2008, Nashville, TN, pp. 26–33 (2008)
38. Koch, N., Knapp, A., Zhang, G., Baumeister, H.: UML-Based Web Engineering: An Approach Based on Standards. In: Web Engineering: Modelling and Implementing Web Applications. Human-Computer Interaction Series, vol. 12, pp. 157–191. Springer, Heidelberg (2008)